MineThrottle: Defending against Wasm In-Browser Cryptojacking

Weikang Bian
Chinese University of Hong Kong
wkbian@cse.cuhk.edu.hk

Wei Meng
Chinese University of Hong Kong
wei@cse.cuhk.edu.hk

Mingxue Zhang
Chinese University of Hong Kong
mxzhang@cse.cuhk.edu.hk

ABSTRACT

In-browser cryptojacking is an urgent threat to web users, where an attacker abuses the users' computing resources without obtaining their consent. In-browser mining programs are usually developed in WebAssembly (Wasm) for its great performance. Several prior works have measured cryptojacking in the wild and proposed detection methods using static features and dynamic features. However, there exists no good defense mechanism within the user's browser to stop the malicious drive-by mining behavior.

In this work, we propose MineThrottle, a browser-based defense mechanism against Wasm cryptojacking. MineThrottle instruments Wasm code on the fly to detect mining behavior using block-level program profiling. It then throttles drive-by mining behavior based on a user-configurable policy. Our evaluation of MineThrottle with the Alexa top 1M websites demonstrates that it can accurately detect and mitigate in-browser cryptojacking with both a low false positive rate and a low false negative rate.

CCS CONCEPTS

• **Security and privacy** → **Browser security**; *Malware and its mitigation.*

KEYWORDS

WebAssembly; Cryptojacking; Cryptocurrency mining

ACM Reference Format:
Weikang Bian, Wei Meng, and Mingxue Zhang. 2020. MineThrottle: Defending against Wasm In-Browser Cryptojacking. In *Proceedings of The Web Conference 2020 (WWW '20), April 20–24, 2020, Taipei, Taiwan.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3366423.3380085

1 INTRODUCTION

Traditionally, websites host online ads as the main revenue source for monetizing their "free" services. Mining cryptocurrencies has become an alternative and attractive way to generate revenue because of the rising value of cryptocurrencies [21]. The miner has to solve a computationally difficult puzzle in return for a coin. Some puzzles (*e.g.*, those of BitCoin [4]) are very difficult to solve using a commodity computer and require special dedicated hardware to make a profit. The development of new blockchain algorithms and cryptocurrencies (*e.g.*, CryptoNight [12] and Monero [7]) make commodity CPU mining practical and profitable. The recent advances in web technologies (*e.g.*, WebAssembly (Wasm) [8] and asm.js [3]) further enable efficient in-browser cryptocurrency mining.

Today, websites can leverage their visitors' computing capabilities to mine cryptocurrency cooperatively. They can place a code snippet in the visitors' browsers, which then join a distributed mining pool operated by the website owners or some other dedicated brokers. The mining code is often implemented using Wasm, which has been recently supported by all mainstream browsers [10, 14, 16, 24], for its great computational efficiency. However, a malicious website may not obtain any consent from the users before placing the mining code. Such malicious practice is called as drive-by cryptocurrency mining, or cryptojacking [11].

Cryptojacking has become an emerging threat to web users. According to a recent report in 2018 [2], there had been a 459 percent increase in illicit cryptocurrency mining malware detection since 2017. Many studies have shown that website owners deploy cryptocurrency mining code for extra profit [9, 15, 28, 30]. Recently, researchers have started to investigate cryptojacking in the wild [11, 26] and proposed several detection strategies [17, 19, 20, 29]. Some detection methods leverage community-maintained blacklists [1, 5, 6, 18] that can be included by ad-blocking software. More advanced techniques rely on either static and/or dynamic features (*e.g.*, code signature [17, 19, 20, 26, 29], CPU usage [23], JS callstack snapshot [17], CPU cache performance data [20], network traffic [19, 26], *etc.*) of cryptocurrency mining websites. Most of such studies aim to measure the malicious practices on a large scale. The findings of these studies can help build better blacklists, whose effectiveness depends on timely updates. However, these mechanisms cannot be leveraged directly by the end users. For example, some method [20] would require the administrator privilege, thus may not be suitable for a normal user. Currently, there exists no good defense mechanism that can help the users suppress in-browser cryptojacking in real time.

In this work, we aim to develop a real time defense mechanism against Wasm in-browser cryptojacking. We face several challenges. First, attackers may apply advanced techniques to evade the detection, *e.g.*, by self-throttling, using random servers, or code obfuscation. A blacklist-based or a static feature-based approach would not work well because of the limitation discussed above. Second, as we also aim to automatically mitigate the mining activities in real time, the defense mechanism shall be able to well distinguish benign websites from cryptojacking websites. Third, the defense mechanism itself shall require minimum computing resources. Either a high false positive rate or a high performance overhead can significantly disrupt the user's browsing experience. Finally, the mechanism shall work well for a non-expert user without requiring the administrator privilege.

To solve the above challenges, we develop MineThrottle, a browser-based defense mechanism against Wasm in-browser cryptojacking. MineThrottle detects cryptocurrency mining Wasm programs (more specifically, threads) with just-in-time code instrumentation and light-weight block-level program profiling. Our method is inspired by the observation that Wasm miners heavily leverage Wasm instructions that are distinct from those frequently used by benign programs. This is evident both from the nature of cryptocurrency mining algorithms—repetitive or iterative computation, and from the instruction execution traces we collect in our study. On the one hand, we discover that the execution traces of cryptocurrency mining programs exhibit huge numbers of repeated sequences of instructions. In other words, most of the CPU time is spent by very few code blocks. On the other hand, such patterns are not observed from the traces of non-mining programs such as games or graphics applications.

MineThrottle first detects similar mining related code blocks at the Wasm just-in-time compilation time using a few block-level statistical instruction features that can well distinguish mining related code blocks from other blocks. It then instruments the suspicious code blocks with performance profiling code. The profiling is very light-weight as it is applied at block level instead of instruction level. At runtime, MineThrottle periodically calculates the *effective mining speed* of a Wasm program (thread)—the number of executed instructions per unit CPU time. Only if the measured mining speed is close to the baseline values learned from a set of known mining programs, it labels it as a positive case. MineThrottle then penalizes the mining thread to release the CPU resource that would be occupied by it to other important tasks of the user. We further allow the users to adjust several custom parameters to control the sensitivity of MineThrottle's detection to limit the impact on false positive cases if any.

We implemented a prototype of MineThrottle based on the V8 engine of the Chromium browser. In particular, we modified the V8 Wasm interpreter and compiler to analyze Wasm instruction execution traces and to inject runtime analysis code, respectively. We performed a large scale experiment on the Alexa top 1M websites, and detected 109 true positive Wasm cryptojacking websites, including a few that were not detected by a state-of-the-art detection mechanism. Our evaluation shows that MineThrottle can accurately and effectively suppress cryptojacking, with both a low false positive rate and a low false negative rate.

In summary, we make the following contributions.

- We develop MineThrottle, a browser-based defense mechanism against Wasm in-browser cryptojacking.
- We perform a measurement study on the Alexa top 1M websites and detect 109 positive mining websites.
- We show that MineThrottle is an effective and robust defense mechanism against cryptojacking.

2 PROBLEM STATEMENT

We focus on detecting the *drive-by* in-browser cryptocurrency mining behavior, *i.e.*, cryptojacking, where the mining procedure starts automatically without obtaining the user's consent or without any user action. Further, we aim to *detect* and *defend against* cryptojacking in real time in a user's browser.

We assume a remote web attacker who controls a website that a user may visit. The attacker can include arbitrary JavaScript code and Wasm code in his/her website. He/She uses mainly Wasm code for cryptocurrency mining, and uses JavaScript code for managing the mining tasks and communicating with a remote server.

The attacker may obfuscate the JavaScript code and Wasm code to evade detection. In other words, simply computing a code or file signature is unable to reliably identify a mining script. Methods that build a script signature by using the function names are also likely to fail. Since Wasm code obfuscation is not widely observed yet, we assume the attacker can perform some basic obfuscation operations, including inserting dummy instructions at random locations. In addition, the attacker may self-throttle mining to circumvent detection methods that rely on observing a high CPU usage. Finally, the attacker may communicate with his/her own custom servers instead of the servers of publicly known mining pools to bypass detection methods using community-maintained public blacklists (*e.g.*, the NoCoin ad block list [18]).

3 METHODOLOGY

In this section, we present our methodology to defend against in-browser cryptojacking. We first present our strategies on identifying mining-related code blocks (§3.1), then discuss how to detect (§3.2) and suppress (§3.3) Wasm cryptojacking, respectively.

3.1 Identifying Mining-related Wasm Code

We seek to understand the behavioral or semantic difference between cryptocurrency mining programs and other programs. In order to perform cryptocurrency mining, a Wasm program usually exhibits the activities of repeating certain computation. In other words, some specific sequences of Wasm instructions would be executed much more frequently on the cryptojacking websites than other websites. This is evident in a prior study [29] which finds that mining programs exhibited different top Wasm instructions than other programs in their profiling, and each mining program may have a distinguishing distribution of top instructions.

We could implement a similar instruction-level profiler at runtime to identify cryptojacking websites. However, this approach would not work in practice for the following reasons. First, an attacker can inject dummy code (*e.g.*, many `i32.and` instructions) to deviate the instruction distribution from the reference ones to bypass detection. Second, profiling at instruction level is not practical. We could leverage some advanced hardware performance monitoring technologies (*e.g.*, Intel Processor Tracing [25])) to profile the programs. However, such advanced features may require the root privilege and they are not necessarily available on all consumer computers. We could also profile a Wasm instruction using additional Wasm code, with an extremely high overhead—four additional instructions are needed to count one.

To overcome the above problems, we extract block-level features that can represent mining activities, *i.e.*, the hash-like operations. We compare the Wasm instruction execution traces of several different applications, including several known cryptocurrency mining websites (*e.g.*, CoinHive and CryptoLoot), with a modified Chromium browser. We count each unique basic block

Table 1: Number of unique top basic blocks of Wasm applications.

CoinHive	CryptoLoot	LightMiner	Crypto Webminer	Coin Web Mining
3	4	10	4	3
Tanks	Wasm Astar	Wasm Asteroids	Qt Hello Window	Wasm Vim
1541	82	63	278	45

in the collected trace, and find the top blocks that consumed accumulatively 90% of the CPU time, which is approximated by the number of instructions of each basic block.

Not surprisingly, on the one hand, the bulk of computation resource was spent by several top blocks in mining programs. On the other hand, there exists almost no block that consumed a large amount (> 5%) of CPU resource in the other programs. In particular, we can find more than 50 distinct code blocks in the top 90% blocks of non-mining programs (*e.g.*, games, graphics), whereas there are less than 10 distinct code blocks in the top 90% blocks of mining programs. We present the number of unique top basic blocks of different Wasm applications in Table 1. We compute the frequency distribution of Wasm instructions in the top basic blocks, and find eight top instructions—the *discriminating instructions*—that can discriminate between top mining and non-mining blocks.

Therefore, we use both the ratio of discriminating instructions in a basic block and the size of a basic block to identify cryptocurrency mining-related basic blocks. We set a threshold for the discriminating instructions identified in a basic block by learning a linear model from the top blocks we identify previously. We report a positive match if the calculated ratio excluding NOP instructions of a basic block is greater than the threshold. However, an attacker may bypass this detection by injecting a lot of dummy code. We argue that this would increase the mining cost of the attacker and lowers the mining efficiency. Nevertheless, to defend against this kind of attacks, we use a lower ratio threshold for larger basic blocks to tolerate the extra injected instructions. In addition, attackers might break a large basic block into several smaller ones to bypass our detection. Thus, we use the absolute number of discriminating instructions to detect small mining blocks.

Although we might label correctly some positive top mining blocks, our method can have false positives. It is likely that a non-mining program would require to perform some operations that are similar to those of the top mining blocks, *e.g.*, calculating a hash value. However, mislabels would not be problematic if they are rare at runtime. We discuss next how we check the frequency that mining-related blocks execute at runtime.

3.2 Detecting Cryptojacking

We check at runtime if the CPU time of one Wasm program (more precisely, a thread) is spent mostly in mining-related code blocks. Measuring the CPU time of every code block would be difficult and unnecessary. Instead, we first compute a baseline average CPU usage that represents the average mining speed of the sample mining programs. Then at runtime, we compute the same metric and compare it with the baseline value.

We measure the runtime CPU usage of mining-related code blocks by performing *block-level profiling*. For each identified positive block, we dynamically instrument it at the just-in-time compilation time to insert profiling code. Since the mining-related basic blocks typically are very large, the runtime overhead caused by our profiling code would be almost negligible and significantly lower than that of instruction-level profiling. Specifically, the profiling code would increment a global performance counter by the number of instructions in the corresponding block each time it executes. By periodically checking the performance counter over a fixed *detection interval*, we can estimate how many mining-related instructions have been executed since the last check. Note that the detection interval needs to be measured in *CPU time* instead of wallclock time, because a thread may share the same CPU core with many other threads/processes. Dividing the increment in the last detection interval by the interval gives us a temporal mining speed of the thread. It represents how much computation is performed by the mining-related code blocks given a fixed amount of CPU resource (time).

To capture potential variation of the mining activities, we measure the temporal mining speeds of true-positive Wasm mining programs in many intervals. We then compute a baseline average speed—S—that represents the average mining speed of mining programs. We also obtain the standard deviation σ when calculating the baseline average speed. Note that for processors with different frequencies, the baseline average speed needs and can be adjusted accordingly. At runtime, we would compute in the same interval a mining speed—*speed*—of a Wasm thread and compare it with the baseline speed. We then would make a *negative* decision if the mining speed is much smaller than the baseline speed. This avoids reporting legitimate Wasm programs that employ very limited amount of mining-related operations as a false positive. To tolerate slower mining threads or to detect mining threads that self-throttle, we use a parameter T_σ to control the absolute detection distance from the baseline speed. A positive decision is made if the following inequality is true. On the one hand, a larger threshold would be more difficult to bypass but may also cause a lot of false positives. On the other hand, a smaller threshold may result in a high false negative rate. We will evaluate the threshold in §5.2.

$$speed \geq S - T_\sigma \sigma \qquad (1)$$

3.3 Defending against Cryptojacking

To preserve the CPU resource of a victim user's machine from cryptojacking, we need to stop the mining activities. However, killing the mining worker thread in the browser may lead to runtime errors, in particular if that thread performs other tasks in addition to controlling a Wasm mining module. Further, in the case of a false positive decision, we could disrupt the normal operation of a legitimate website.

To reduce the impact of false positives, we suspend a positive thread as a penalty, instead of abruptly killing it. If it were a true positive, the mining activity would be effectively stopped temporarily in a *sleep interval*. We would be able to detect it again after it resumes and continues mining. On the contrary, if it were a false positive, we would cause only a delay in completing its jobs instead of disrupting it. A delay would usually be expected because the worker thread may share the processor with many other threads and processes. To avoid a significant delay, a user can configure a smaller sleep interval. We will discuss how this parameter affect's the performance of our defense mechanism in §6.1.

Next, we depict how we implement our methodology in our system—MineThrottle—a browser-based defense mechanism against drive-by Wasm cryptocurrency mining.

4 MINETHROTTLE

In this section, we present the design and implementation of MineThrottle, a system that can throttle Wasm in-browser cryptojacking in real time. MineThrottle can operate in three modes. In the *interpreting mode*, it can log all Wasm instructions that are interpreted to help study the semantic features of Wasm programs. In the *detection mode*, it performs just-in-time Wasm code instrumentation and block-level program profiling to detect mining-related threads. In the *defense mode*, it further penalizes the mining-related threads to preserve CPU resources. We next describe our prototype implementation of MineThrottle based on the V8 JavaScript engine of the Chromium browser (version 71.0.3578.98).

4.1 Interpreting Mode

The Chromium V8 engine can both compile and interpret Wasm code. To obtain the Wasm instruction execution trace, we configure the V8 engine to execute the Wasm instructions in its interpreter. We further hook the V8 Wasm interpreter to inspect the Wasm bytecode that is being interpreted one by one. Specifically, MineThrottle hooks the method `ThreadImpl::Execute()` of the interpreter to inspect all the executed Wasm instructions and dump the opcode stream into a local file for further analysis. With the dumped execution trace, we can identify top instructions, top subsequences of instructions, and top basic blocks, as we had discussed in §3.

4.2 Detection Mode

MineThrottle performs just-in-time Wasm code instrumentation for block-level program profiling and detection of cryptocurrency mining-related threads. We modify the V8 Wasm compiler to insert our custom profiling and detection code.

The basic executable unit of Wasm code is a module, which is structured into multiple functional sections, *e.g.*, the type, function, global and code sections, *etc.* To profile the mining-related basic blocks, we inject one extra global performance counter into the global section of each module. The V8 Wasm compiler compiles the modules at the granularity of functions. Therefore, we hook the V8 method for decoding Wasm function body to identify mining-related basic blocks. When it detects a basic block that is mining-related, according to our method in §3.2, MineThrottle injects a trigger at the end of the basic block. The trigger updates the global performance counter by the size of the corresponding basic block.

For each module, we further define an extra `module_info` structure, which contains the id of the thread that is executing the module, the CPU time spent by the thread and the value of the global performance counter since last check. We then further register a custom signal handler within the module compiler to periodically inspect the current cumulative CPU usages and the performance counters of all Wasm threads. With these performance data, MineThrottle can calculate the mining speed of each thread. Specifically, it would check for each Wasm thread if the *CPU time* difference since the last update is greater than the *detection interval*. If true, it calculates the mining speed by dividing the difference in the two performance counter values by the elapsed CPU time. It then updates the performance data in the thread's `module_info` with the latest measurement. Next, it checks if the condition specified in

Table 2: Mining speeds of sample cryptojacking websites.

Benchmark	Detection Interval (ms)			
	100	250	500	1000
Average	2,118,789	2,276,725	2,275,011	2,277,306
Std-Dev.	1,069,980	327,670	280,611	182,818

Equation 1 is met to label the current thread as performing mining-related operations in the last detection interval.

4.3 Defense Mode

In the defense mode, MineThrottle can further penalize a suspicious mining-related thread to release occupied computing resources. Specifically, it checks the corresponding thread ID and sends a customized signal `SIGUSR1` to the target thread. The signal handler, which is registered in the module compiler as well, calls the `sleep()` function to put the mining thread into sleep for the duration of *sleep interval* seconds. The sleep interval can be configured by the user and can also be adjusted dynamically depending on the CPU resource availability of the system and the amount of CPU time that has been consumed by the thread and the process.

The above procedure keeps running until the execution of the Wasm program completes. In other words, when the mining thread becomes active again, it could still be detected and penalized by MineThrottle. In this way, we can effectively slow down the mining threads as a defense against cryptojacking, without impacting the execution of other normal threads. Further, the runtime overhead caused by MineThrottle would be negligible for benign programs not performing many mining-related operations, because our profiling code will not be executed frequently. Even for a true positive mining program the overhead is still low, because we perform block-level profiling instead of instruction-level profiling.

5 DETECTION EVALUATION

In this section, we evaluate if MineThrottle can accurately detect Wasm in-browser cryptojacking in the real world.

5.1 Experiment Setup and Data

We performed a large-scale experiment by visiting the main pages of the Alexa top 1M websites in September, 2019. We configured the MineThrottle prototype to log the performance counters and other necessary data of each website (frame) in a file. We discovered 659 websites that employed Wasm code in our experiment. We then manually visited each Wasm-related website for up to 3 times to verify it was indeed performing drive-by cryptocurrency mining. In total, we found 109 cryptojacking websites. Our prototype failed to load 2 cryptojacking websites correctly because of some unknown implementation bug. We leave it as a future engineering work.

5.2 Detection Result

We adjusted the parameters of our detection algorithm to evaluate its performance. We evaluated four different values (100 ms, 250 ms, 500 ms, and 1000 ms) for the detection interval. We randomly selected 5 top mining websites to train the detection model. The baseline averages and standard deviations of the mining speeds learned from the top websites are shown in Table 2.

The average mining speeds in different detection intervals are quite close. However, the standard deviation increased significantly

Table 3: Detection performance evaluation of MineThrottle.

Benchmark	MineThrottle T_σ											
	0.3				1				3			
	Detection Interval (ms)				Detection Interval (ms)				Detection Interval (ms)			
	100	250	500	1000	100	250	500	1000	100	250	500	1000
FP (#)	0	0	0	0	2	0	0	0	47	0	0	0
FPR (%)	0	0	0	0	0.36	0	0	0	8.55	0	0	0
FN (#)	2	2	2	13	2	2	2	11	2	2	2	2
FNR (%)	1.83	1.83	1.83	11.93	1.83	1.83	1.83	10.09	1.83	1.83	1.83	1.83

with the drop of detection interval. Therefore, a large detection interval would be preferred for reliable detection. Otherwise, we need a small T_σ for small detection intervals. In our experiment, we set T_σ to 0.3, 1, and 3, respectively.

We report one website as mining only if Equation 1 is true for at least 5 times of a thread. The evaluation result is shown in Table 3. There are always 2 false negative (FN) cases that our prototype failed to render when the detection interval is less than 1000 ms. Excluding the two websites, we would have a 0 false-negative rate (FNR). However, we have more FN cases when the detection interval is 1000 ms, because the standard deviation is the smallest. Therefore, many speeds may fall under the threshold, particularly when T_σ is smaller than 3.

We observed some false positive (FP) cases when the detection interval is 100 ms, where the standard deviation is almost 50% of the baseline average. Given a large T_σ, *e.g.*, 1 or 3, many benign threads may be classified as positive because their mining speeds can easily exceed the threshold. In particular, when T_σ is greater than 2, Equation 1 is always true. We had only 47 false positive cases because only they contained basic blocks that were detected as mining related. For the rest websites, our profiling code was never injected.

5.2.1 Comparison with Other Tools. We compared MineThrottle with a state-of-the-art cryptocurrency mining detection tool – MineSweeper [20], which detects the mining behavior based on static Wasm code analysis and runtime CPU performance measurement data. We also tried to use other recent detection tools like Outguard [19] and CMTracker [17]. We were not able to use their released code to generate valid data for analysis.

We ran MineSweeper with its default setting to check the same 659 websites for 5 times. MineSweeper achieved a 0.54% FPR and a 27.10% FNR. MineThrottle was able to detect 29 extra positive cases, and all the positive cases detected by MineSweeper except for the 2 that it failed to load. MineSweeper always failed to load 13 positive cases, and labeled the other 16 as negative cases because they either implemented different functions not fingerprinted by MineSweeper or started mining after MineSweeper's profiling period. MineSweeper detected 3 false positive cases, because all of them failed to establish a WebSocket connection to remote mining servers that were shut down. Our evaluation shows that MineThrottle can accurately detect true mining activities.

6 DEFENSE EVALUATION

In this section, we first demonstrate how MineThrottle can effectively throttle such unwanted mining behaviors (§6.1). Next, we evaluate the performance impact on benign websites that do not perform cryptocurrency mining using Wasm (§6.2). All experiments are run on a Debian 4.19 desktop machine equipped with an Intel Core i7-8700K processor and 64GB RAM.

Table 4: Performance measurement when evaluating sleep interval and detection interval.

Website	Benchmark	MineThrottle Sleep Interval (s)								
		Vanilla	5				10			
			Detection Interval (ms)				Detection Interval (ms)			
			100	250	500	1000	100	250	500	1000
filmaidykai.net	System CPU (%)	96.15	3.50	7.19	10.88	18.79	2.31	4.23	6.14	11.12
	Browser CPU (%)	95.35	2.61	6.32	10.11	17.91	1.45	3.28	5.31	10.29
300mbfilms.co	System CPU (%)	84.23	4.05	6.21	10.32	16.16	2.22	3.66	5.70	9.74
	Browser CPU (%)	83.43	3.16	5.33	9.54	15.48	1.29	2.88	5.03	8.73
browsermine.com	System CPU (%)	81.10	3.43	7.13	12.98	20.38	2.55	4.43	6.66	11.70
	Browser CPU (%)	80.55	2.74	6.48	12.32	19.52	1.97	3.78	6.02	11.04
seriesf.lv	System CPU (%)	40.52	3.91	7.16	10.40	15.87	2.65	4.71	6.72	10.49
	Browser CPU (%)	39.56	2.98	6.27	9.61	15.23	1.82	3.76	5.83	9.64

6.1 Cryptojacking Websites

We evaluate how MineThrottle can suppress drive-by Wasm cryptocurrency mining. We select four mining websites that can be detected by MineThrottle. We use both MineThrottle and a vanilla Chromium browser of the same version to visit each website for at least 60 seconds. We collect the whole system CPU usage measured as %CPU during the visit using the tool `mpstat`. In the meanwhile, we also monitor the CPU usages of the browser process and its threads using the tool `pidstat`. The collected CPU usages are normalized by the two tools to a maximum value of 100%.

We evaluate four values (100 ms, 250 ms, 500 ms, and 1000 ms) for the detection interval, which is the CPU-time interval that MineThrottle uses to measure how much CPU resource a thread spends on executing code related to cryptocurrency mining in the last window (interval). We evaluate two values (5 seconds, and 10 seconds) for the sleep time, for which MineThrottle suspends a suspicious mining thread. We set T_σ to 1.

We present the results in Table 4. The four mining websites consumed from 39.56% up to 95.35% CPU of all cores on average with the vanilla Chromium browser. MineThrottle effectively throttled the drive-by mining activities when visiting all the four mining websites. In particular, when configured with a 10-second sleep interval and 100-millisecond CPU-time detection interval, the average browser CPU usage when visiting filmaidykai.net and seriesf.lv plunged from 95.35% to 1.45% and from 39.56% to 1.82%, respectively. We can also observe significant CPU usage drop to 1.29% and 1.97% for 300mbfilms.co and browsermine.com, respectively.

Although we discover a fall in CPU usage for all configurations of MineThrottle, the descents are quite different. First, when the sleep interval is fixed, a larger detection interval would cause a higher CPU usage, because MineThrottle needs to wait for longer in CPU time for each thread to make a decision. This gives the mining threads more time to execute, resulting higher CPU usages. On the contrary, a smaller detection interval would allow MineThrottle to detect cryptojacking earlier. Second, when the detection interval is fixed, a larger sleep interval would preserve more CPU resources for the other legitimate tasks running on the user's machine. Setting a large sleep interval is desired if the CPU power is scarce with respect to the workload. However, if a non-mining thread is mistakenly detected as mining (a false positive), a large sleep interval would prevent it from completing its jobs promptly. Thanks to MineThrottle's strong ability of accurately fingerprinting

Table 5: Performance measurement when visiting lordz.io.

Benchmark	Vanilla	MineThrottle Detection Interval (ms) 100	250	500	1000
System CPU (%)	12.61	12.34	13.27	12.70	13.12
Renderer-Process CPU (%)	5.03	4.77	5.08	5.08	4.96
GPU-Process CPU (%)	2.55	2.63	2.58	2.57	2.57

Table 6: Performance measurement when visiting google.com.

Benchmark	Vanilla	MineThrottle Detection Interval (ms) 100	250	500	1000
System CPU (%)	0.35	0.34	0.31	0.30	0.34
Browser CPU (%)	0.01	0.01	0.01	0.03	0.01

and detecting Wasm mining algorithms, such false positive cases would be very rare, as we had demonstrated in §5.

6.2 Benign Websites

We evaluate the performance overhead that may be incurred due to MineThrottle's periodic detection routine. We show that MineThrottle does not affect the operation of benign websites. We select two benign websites—a non-mining Wasm website lordz.io, and a non-Wasm website google.com—in our evaluation. The T_σ is set to 1. The same performance measurement is collected in the experiment.

Non-mining Wasm Websites. We present in Table 5 the performance measurement result when visiting the website lordz.io, a game website implemented with Wasm and asm.js. With a vanilla Chromium browser, the average system CPU usage was as high as 12.61%. When visited with MineThrottle using different detection intervals, the system CPU usage and the browser CPU usages almost did not vary. This demonstrates that the runtime performance overhead caused by MineThrottle is negligible, because it performs mostly primitive arithmetic calculations at a low frequency (at most once per 100 ms).

We did observe that this website had two threads that included mining-related basic blocks. For all settings of detection interval, MineThrottle did not report any positive thread because the mining speeds calculated were much smaller than its detection threshold. This shows that our algorithm can correctly recognize benign Wasm applications that perform a light amount of mining-related operations, *e.g.*, calculating hash values.

Non-Wasm Websites. We present in Table 6 the performance measurement result when visiting the website google.com, on which we did not detect any Wasm code. MineThrottle did not introduce any observable runtime CPU overhead, because its detection code would not be executed at all on non-Wasm websites.

7 RELATED WORK

Blacklist-based Defense. The most widely used cryptojacking detection and defense mechanisms rely on blacklist filtering [1, 5, 6, 18]. Generally, the blacklists are maintained by communities and are compatible with mainstream ad blocking software and browser extensions. However, this mechanism suffers from both high false positive rate and false negative rate, because the updates on blacklists are usually lagging, and the features can be easily manipulated by attackers.

Static or Dynamic Features Based Detection. MineSweeper proposed to build code signatures upon three cryptographic related instructions to identify cryptocurrency mining code [20]. It also showed that the mining code exhibits higher frequency of data load and store events in CPU cache, which could serve as an alternative feature. However, monitoring CPU cache events requires the administrator privilege, which makes it infeasible to be used as defense for common users. Hong *et al.* designed two types of profilers that capture cryptojacking behaviors [17]. Specifically, the hash based profiler monitors the hash functions with certain fixed names, and the stack structure profiler examines if the same call stack appears periodically. Musch *et al.* combined the continuously high CPU usage with the number of Web Workers and presence of WebAssembly code to detect miners [23]. However, it has been proved that CPU profiling is unreliable, because the attacker can perform CPU throttling to evade the detection [17, 20]. Rüth *et al.* proposed to monitor WebSockets and WebAssembly code as the indicators of cryptojacking [26]. Similarly, Kharraz *et al.* introduced a novel method to detect cryptojacking using an SVM classifier trained on seven distinct features, (*e.g.*, parallel tasks, WebSockets, hash algorithms, *etc.*) [19]. They detected hash algorithms by searching for the specific function names or code signatures, which can be easily evaded by code obfuscation. Moreover, they built the ground truth dataset based on community-maintained blacklists which may have misclassifications. In addition, Wasm analysis has been studied by several works [13, 22, 27].

Semantic-based Detection. Wang *et al.* introduced SEISMIC to detect cryptojacking based on semantic signatures [29]. SEISMIC instruments WebAssembly code to maintain counters of certain WebAssembly instructions. This approach mitigates the effects of basic code obfuscation, but creates high overhead. The idea of SEISMIC is quite similar to ours. However, we use block-level semantic features to detect cryptojacking. Our method introduces much lower overhead and is more robust to obfuscation. We implemented our detection mechanism in the Chromium browser so that we can detect cryptojacking at runtime with minimal human intervention. We also implemented the defense mechanism to suppress mining threads at runtime, which is not supported by most existing detection systems.

8 CONCLUSION

We proposed MineThrottle, a browser-based system, to defend against Wasm in-browser cryptojacking. It accurately detects cryptocurrency mining behavior by performing light-weight basic block level profiling. It then penalizes the mining thread to preserve a victim user's CPU resource. MineThrottle achieved a 0% false positive rate and a 1.83% false negative rate in our evaluation with the Alexa TOP 1M websites, and was able to effectively throttle the drive-by mining activities.

ACKNOWLEDGMENTS

The work described in this paper was partly supported by a grant from the Research Grants Council of the Hong Kong Special Administrative Region, China (CUHK 24209418).

REFERENCES

[1] 2018. Dr. Mine. https://github.com/1lastBr3ath/drmine.

[2] 2018. The Illicit Cryptocurrency Mining Threat. https://www.cyberthreatalliance.org/wp-content/uploads/2018/09/CTA-Illicit-CryptoMining-Whitepaper.pdf.

[3] 2019. asm.js. http://asmjs.org/.

[4] 2019. Bitcoin. https://bitcoin.org/.

[5] 2019. CoinBlocker Lists. https://zerodot1.gitlab.io/CoinBlockerListsWeb.

[6] 2019. MinerBlock. https://github.com/xd4rker/MinerBlock.

[7] 2019. Monero. https://www.getmonero.org.

[8] 2019. WebAssembly. https://webassembly.org/.

[9] R. Neumann an A. Toro. April 2018. In-browser mining: Coinhive and WebAssembly. https://blogs.forcepoint.com/security-labs/browser-mining-coinhive-and-webassembly.

[10] J. DeMocker. November 2017. WebAssembly support now shipping in all major browsers. https://blog.mozilla.org/blog/2017/11/13/webassembly-in-browsers/. In *Mozilla Blog*.

[11] Shayan Eskandari, Andreas Leoutsarakos, Troy Mursch, and Jeremy Clark. 2018. A first look at browser-based Cryptojacking. In *2018 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*. London, UK.

[12] Seigen et al. March 2013. CryptoNight Hash Function. https://cryptonote.org/cns/cns008.txt.

[13] William Fu, Raymond Lin, and Daniel Inge. 2018. TaintAssembly: Taint-Based Information Flow Control Tracking for WebAssembly. *arXiv preprint arXiv:1802.01050* (2018).

[14] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. 2017. Bringing the Web up to Speed with WebAssembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Barcelona, Spain.

[15] Alex Hern. September 2017. Ads don't work so websites are using your electricity to pay the bills'. https://www.theguardian.com/technology/2017/sep/27/pirate-bay-showtime-ads-websites-electricity-pay-bills-cryptocurrency-bitcoin.

[16] David Herrera, Hangfen Chen, Erick Lavoie, and Laurie Hendren. 2018. *WebAssembly and JavaScript Challenge: Numerical program performance using modern browser technologies and devices.* Technical Report. SABLE-TR-2018-2. Montréal, Québec, Canada: Sable Research Group, School of Computer Science, McGill University.

[17] Geng Hong, Zhemin Yang, Sen Yang, Lei Zhang, Yuhong Nan, Zhibo Zhang, Min Yang, Yuan Zhang, Zhiyun Qian, and Haixin Duan. 2018. How you get shot in the back: A systematical study about cryptojacking in the real world. In *Proceedings of the 2018 ACM Conference on Computer and Communications Security (CCS)*. Toronto, Canada.

[18] Hosh (hoshsadiq). 2019. Github: Block lists to prevent JavaScript miners. https://github.com/hoshsadiq/adblock-nocoin-list.

[19] Amin Kharraz, Zane Ma, Paul Murley, Charles Lever, Joshua Mason, Andrew Miller, Nikita Borisov, Manos Antonakakis, and Michael Bailey. 2019. Outguard: Detecting In-Browser Covert Cryptocurrency Mining in the Wild. In *Proceedings of the Web Conference (WWW)*. San Francisco, CA.

[20] Radhesh Krishnan Konoth, Emanuele Vineti, Veelasha Moonsamy, Martina Lindorfer, Christopher Kruegel, Herbert Bos, and Giovanni Vigna. 2018. Minesweeper: An in-depth look into drive-by cryptocurrency mining and its defense. In *Proceedings of the 2018 ACM Conference on Computer and Communications Security (CCS)*. Toronto, Canada.

[21] Hon Lau. 2017. Browser-based cryptocurrency mining makes unexpected return from the dead. *Sympantec Threat Intelligence* (2017).

[22] Daniel Lehmann and Michael Pradel. 2019. Wasabi: A Framework for Dynamically Analyzing WebAssembly. In *Proceedings of the 24th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Providence, RI.

[23] Martin Johns Marius Musch, Christian Wressnegger and Konrad Rieck. 2018. Web-based Cryptojacking in the Wild. *arXiv preprint arXiv:1808.09474v1 [cs.CR]* (2018).

[24] Marius Musch, Christian Wressnegger, Martin Johns, and Konrad Rieck. 2019. New Kid on the Web: A Study on the Prevalence of WebAssembly in the Wild. In *Proceedings of the 16th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment.* Gothenburg, Sweden.

[25] James R. September 2013. Processor Tracing. https://software.intel.com/en-us/blogs/2013/09/18/processor-tracing.

[26] Jan Rüth, Torsten Zimmermann, Konrad Wolsing, and Oliver Hohlfeld. 2018. Digging into browser-based crypto mining. In *Proceedings of the ACM Internet Measurement Conference (IMC)*. Boston, MA.

[27] Aron Szanto, Timothy Tamm, and Artidoro Pagnoni. 2018. Taint Tracking for WebAssembly. *arXiv preprint arXiv:1807.08349* (2018).

[28] TrendMicro. January 2018. Malvertising Campaign Abuses Google's DoubleClick to Deliver Cryptocurrency Miners. https://blog.trendmicro.com/trendlabs-security-intelligence/malvertising-campaign-abuses-googles-doubleclick-to-deliver-cryptocurrency-miners/.

[29] Wenhao Wang, Benjamin Ferrell, Xiaoyang Xu, Kevin W Hamlen, and Shuang Hao. 2018. SEISMIC: SEcure in-lined script monitors for interrupting cryptojacks. In *Proceedings of the 23rd European Symposium on Research in Computer Security (ESORICS)*. Barcelona, Spain.

[30] Mark Ward. October 2017. Websites hacked to mint crypto-cash. http://www.bbc.com/news/technology-41518351.